

Concurrency

Motivation

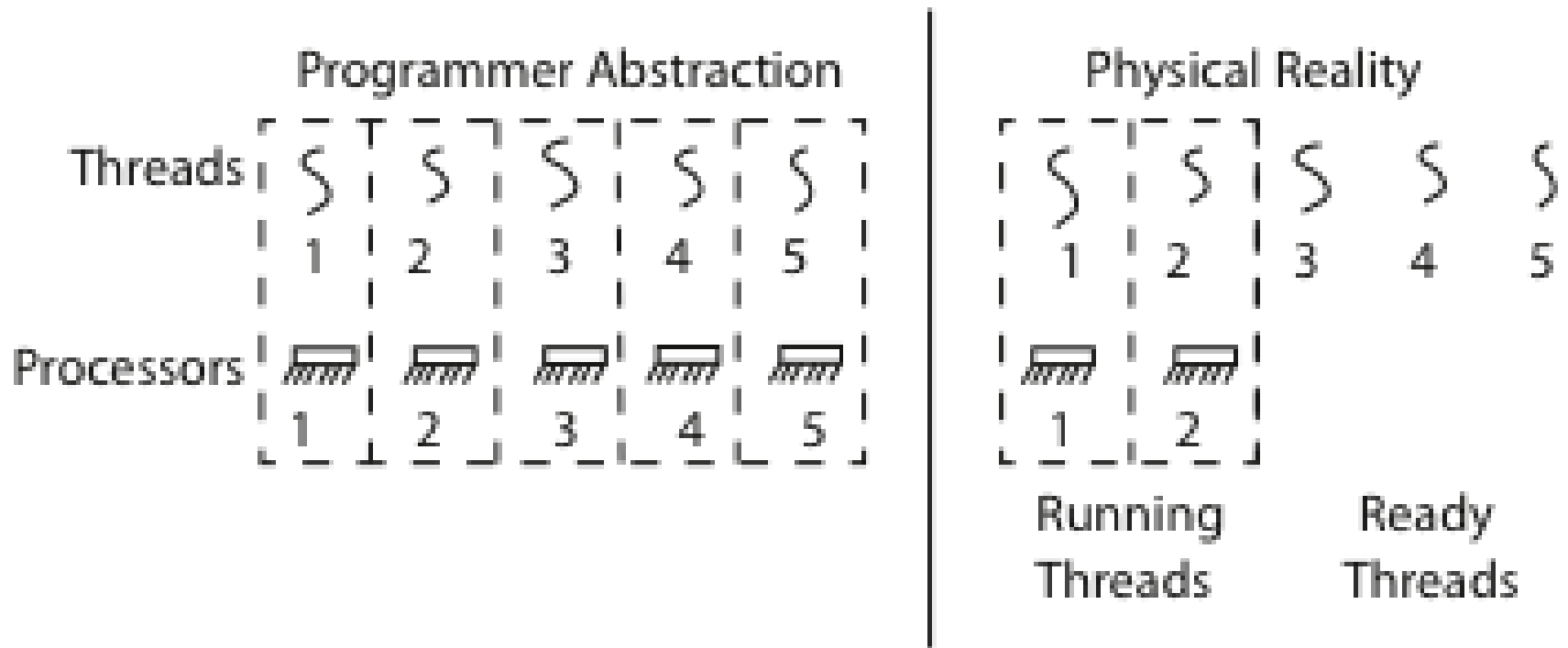
- Operating systems need to be able to handle multiple things at once (MTAO)
 - processes, interrupts, background system maintenance
- Servers need to handle MTAO
 - Multiple connections handled simultaneously
- Parallel programs need to handle MTAO
 - To achieve better performance
- Programs with user interfaces often need to handle MTAO
 - To achieve user responsiveness while doing computation
- Network and disk bound programs need to handle MTAO
 - To hide network/disk latency

Definitions

- A thread is a single execution sequence that represents a separately schedulable task
- Protection is an orthogonal concept
 - Can have one or many threads per protection domain
 - Single threaded user program: one thread, one protection domain
 - Multi-threaded user program: multiple threads, sharing same data structures, isolated from other user processes
 - Multi-threaded kernel: multiple threads, sharing kernel data structures, capable of using privileged instructions

Thread Abstraction

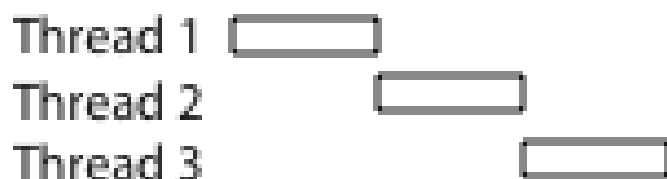
- Infinite number of processors
- Threads execute with variable speed
 - Programs must be designed to work with any schedule



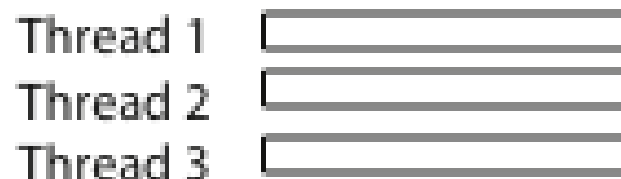
Programmer vs. Processor View

Programmer's View	Possible Execution #1	Possible Execution #2	Possible Execution #3
.	.	.	.
.	.	.	.
.	.	.	.
$x = x + 1;$	$x = x + 1;$	$x = x + 1$	$x = x + 1$
$y = y + x;$	$y = y + x;$	$y = y + x$
$z = x + 5y;$	$z = x + 5y;$	thread is suspended
.	.	other thread(s) run	thread is suspended
.	.	thread is resumed	other thread(s) run
.	thread is resumed
		$y = y + x$
		$z = x + 5y$	$z = x + 5y$

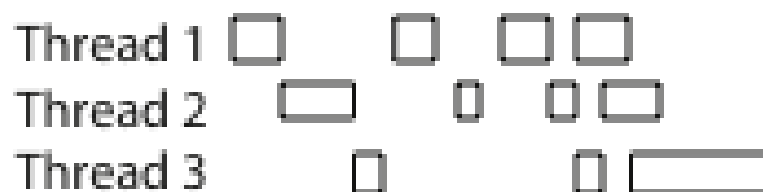
Possible Executions



a) One execution



b) Another execution



c) Another execution

Thread Operations

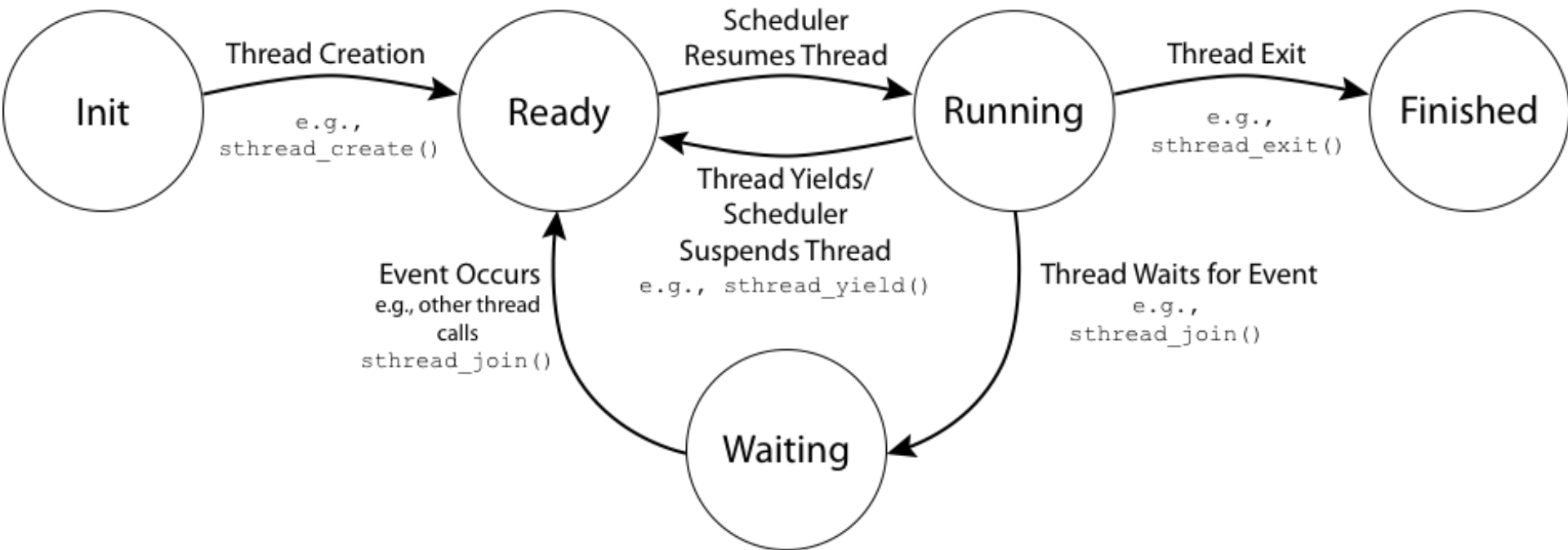
- `sthread_fork(func, args)`
 - Create a new thread to run `func(args)`
 - Pintos: `thread_create`
- `sthread_yield()`
 - Relinquish processor voluntarily
 - Pintos: `thread_yield`
- `sthread_join(thread)`
 - In parent, wait for forked thread to exit, then return
 - Pintos: tbd (see section)
- `sthread_exit`
 - Quit thread and clean up, wake up joiner if any
 - Pintos: `thread_exit`

Main: Fork 10 threads call join on them, then exit

- What other interleavings are possible?
- What is maximum # of threads running at same time?
- Minimum?

```
bash-3.2$ ./threadHello
Hello from thread 0
Hello from thread 1
Thread 0 returned 100
Hello from thread 3
Hello from thread 4
Thread 1 returned 101
Hello from thread 5
Hello from thread 2
Hello from thread 6
Hello from thread 8
Hello from thread 7
Hello from thread 9
Thread 2 returned 102
Thread 3 returned 103
Thread 4 returned 104
Thread 5 returned 105
Thread 6 returned 106
Thread 7 returned 107
Thread 8 returned 108
Thread 9 returned 109
Main thread done.
```


Thread Lifecycle



Location of thread's per thread state

State of thread	Location of TCB	Location of registers
INIT	Being created	TCB
READY	Ready List	TCB
RUNNING	Running List	Processor
WAITING	Synchronization Variable's Waiting List	TCB
FINISHED	Finished List, then Deleted	TCB

Implementing threads

- Thread_fork(func, args)
 - Allocate thread control block
 - Allocate stack
 - Build stack frame for base of stack (stub)
 - Put func, args on stack
 - Put thread on ready list
 - Will run sometime later (maybe right away!)
- stub(func, args): Pintos switch_entry
 - Call (*func)(args)
 - Call thread_exit()

Shared vs. Per-Thread State

Shared State

Heap

Global Variables

Code

Per-Thread State

Thread Control Block (TCB)

Stack Information

Saved Registers

Thread Metadata

Stack

Per-Thread State

Thread Control Block (TCB)

Stack Information

Saved
Registers

Thread Metadata

Stack

Thread Stack

- What if a thread puts too many procedures on its stack?
 - What should happen?
 - What happens in Java?
 - What happens in Linux?

Roadmap

- Threads can be implemented in any of several ways
 - Multiple user-level threads, inside a UNIX process (early Java)
 - Multiple single-threaded processes (early UNIX)
 - Mixture of single and multi-threaded processes and kernel threads (Linux, MacOS, Windows)
 - To the kernel, a kernel thread and a single threaded user process look quite similar
 - Scheduler activations (Windows)

Processes and Threads representation

- Process Control Block (PCB)
 - Data structure associated to each process
- Process Table
 - Contains all PCBs
 - In the kernel, one for the entire system
- Thread Control Block (TCB)
 - One for each thread
- Thread Table
 - One for each process (user-level threads)
 - In the kernel, one for the entire system (kernel-level threads)

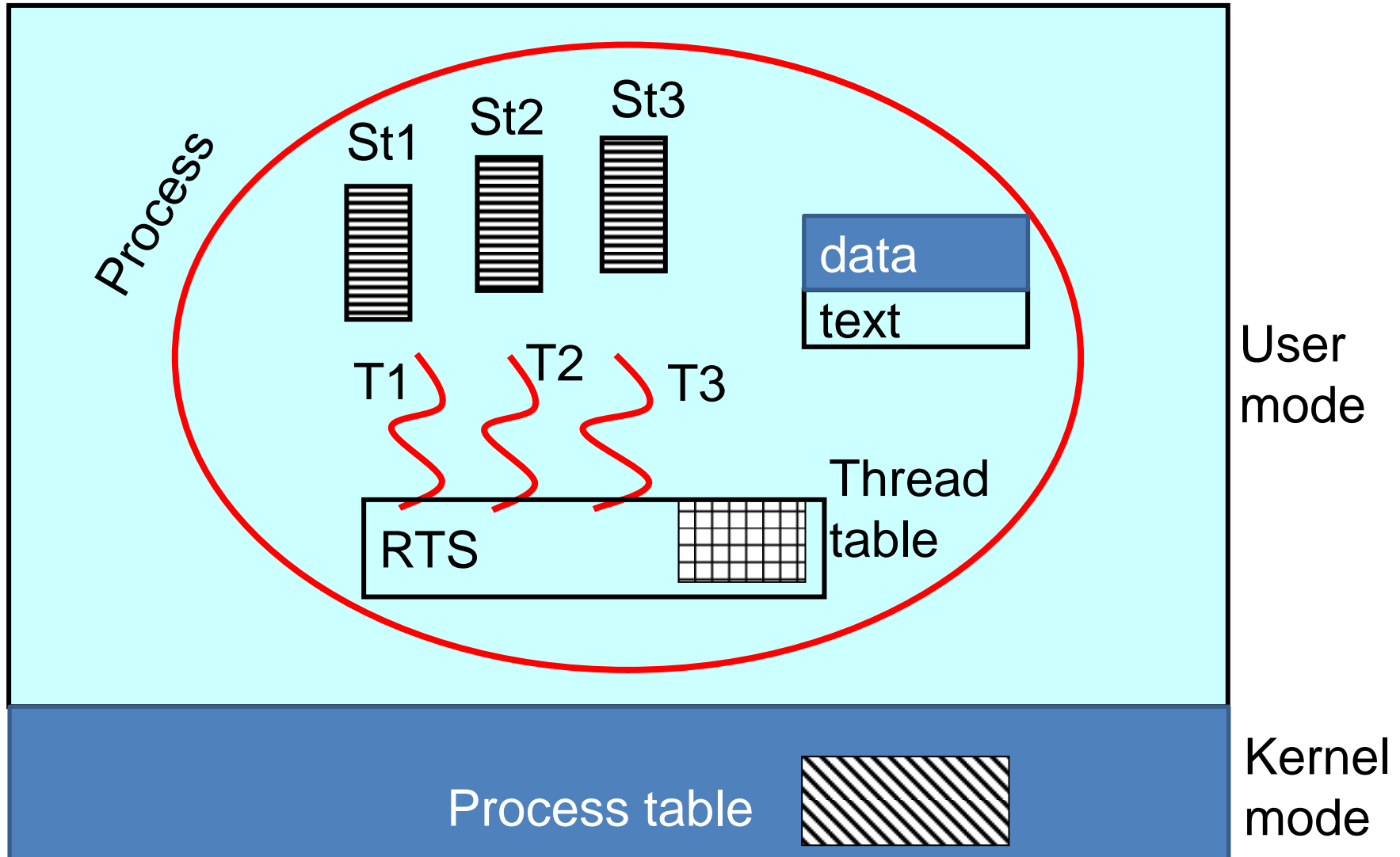
TCB & PCB

- PCB:
 - Process name (PID)
 - Assigned memory
 - Other resources
 - Devices, open files, ...
 - Handlers to the process' threads
 - ...
- TCB:
 - Thread ID
 - State
 - Context of the thread
 - Scheduling parameters
 - Reference to the stack
 - ...

User-level threads

- Threads implemented by means of a user-level library
- O.S. not aware of user level threads
- Thread table within each process
- Scheduling of the threads implemented by the run time support of the process
 - Threads can use *thread_yield()* to release the processor
- An invocation to a blocking system call blocks all the threads

User-level threads



User-level threads

Pros:

- Creation, termination and context switch very efficient
 - Do not need system call invocations, just calls to the thread library
 - In case of context switch the addressing space remains the same
- Can be implemented on any O.S. that does not supports multithreading
 - e.g. early versions of UNIX

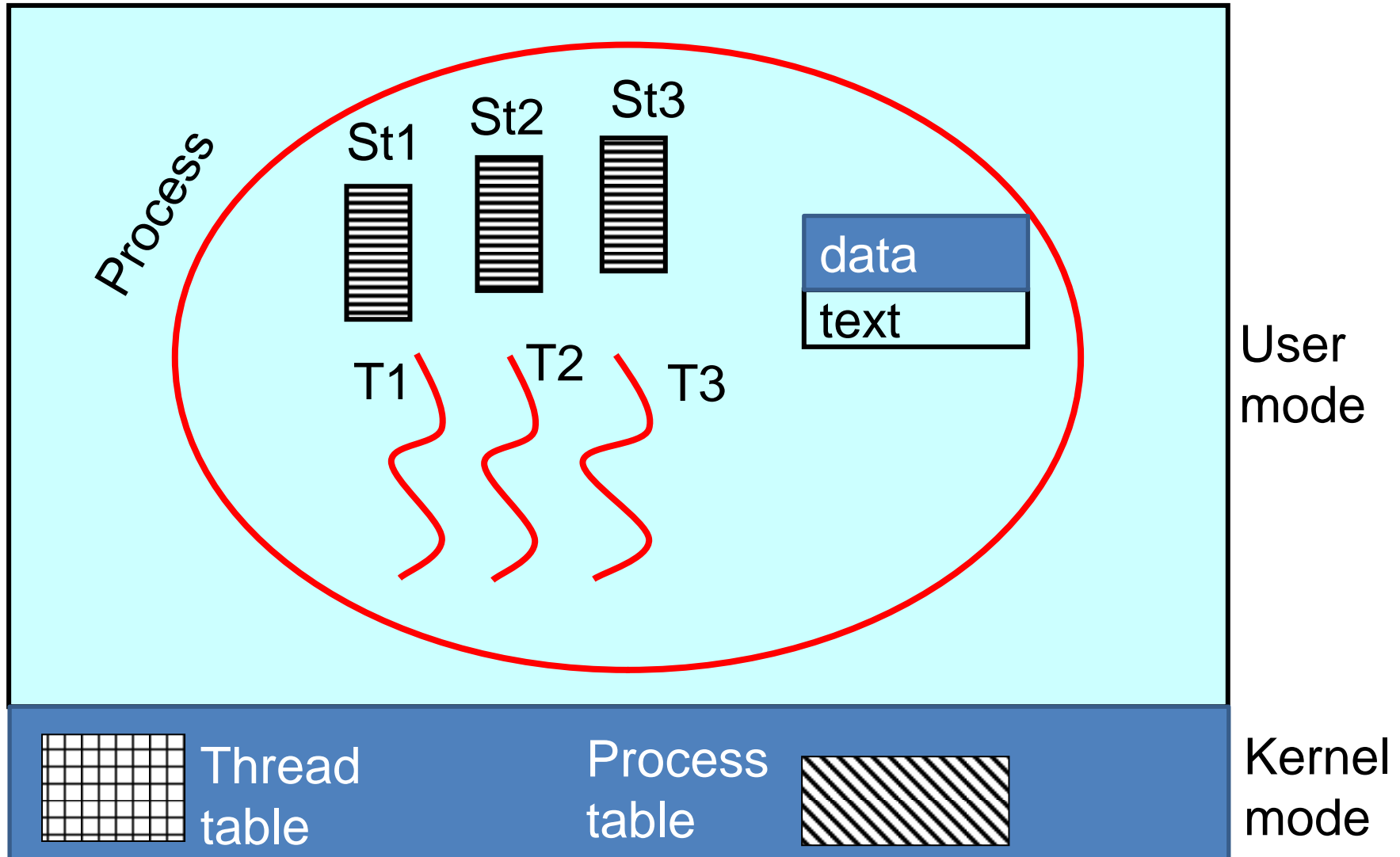
Cons:

- Blocking system calls block all the user-level threads of a process
- Do not take advantage of multiprocessors architectures
 - All threads of a process are scheduled on the same processor

Kernel-level threads

- Threads implemented in the kernel
 - Thread table in the kernel
 - Creation, termination and context switch activated by system calls
 - Different thread of the same process can run in parallel on different processors

Kernel-level threads



Kernel-level threads

- Operations on threads and interactions among threads by means of system calls
 - More overhead w.r.t user-level threads
- Thread scheduling implemented by the O.S.
- Threads can invoke blocking system calls
 - Only the invoker gets blocked

Threads in a Process

- Threads are useful at user-level
 - Parallelism, hide I/O latency, interactivity
- Option A (early Java): user-level library, within a single-threaded process
 - Library does thread context switch
 - Kernel time slices between processes, e.g., on system call I/O
- Option B (Linux, MacOS, Windows): use kernel threads
 - System calls for thread fork, join, exit (and lock, unlock,...)
 - Kernel does context switching
 - Simple, but a lot of transitions between user and kernel mode
- Option C (Windows): scheduler activations
 - Kernel allocates processors to user-level library
 - Thread library implements context switch
 - System call I/O that blocks triggers upcall
- Option D: Asynchronous I/O

Thread switch

- Two causes:
 - Voluntary
 - Due to an interrupt/exception
- Almost the same management for the different cases
 - Kernel/user threads
 - Multithread/singlethread processes

Implementing (voluntary) thread context switch

- User-level threads in a single-threaded process
 - Save registers on old TCB
 - Switch to new stack, new thread
 - Restore registers from new thread's TCB
 - Return
- Kernel threads
 - Exactly the same!
 - Pintos: thread switch always between kernel threads, not between user process and kernel thread

Pintos: switch_threads (oldT, nextT) (interrupts disabled!)

```
# Save caller's register state
# NOTE: %eax, etc. are ephemeral
# This stack frame must match the
  one set up by thread_create()
pushl %ebx
pushl %ebp
pushl %esi
pushl %edi

# Get offset of (struct thread, stack)
mov thread_stack_ofs, %edx
# Save current stack pointer to old
  thread's stack, if any.
movl SWITCH_CUR(%esp), %eax
movl %esp, (%eax,%edx,1)
```

```
# Change stack pointer to new
  thread's stack
# this also changes currentThread
movl SWITCH_NEXT(%esp), %ecx
movl (%ecx,%edx,1), %esp

# Restore caller's register state.
popl %edi
popl %esi
popl %ebp
popl %ebx
ret
```

Two threads call yield

Thread 1's instructions

call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state

return thread_yield
call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state

return thread_yield

...

Thread 2's instructions

call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state

return thread_yield
call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state

...

Processor's instructions

call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state
call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state
return thread_yield
call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state
return thread_yield
call thread_yield
save state to stack
save state to TCB
choose another thread
load other thread state
return thread_yield

...

Thread switch on an interrupt

- Thread switch can occur due to timer or I/O interrupt
 - Tells OS some other thread should run
- Simple version (Pintos)
 - End of interrupt handler calls `switch_threads()`
 - When resumed, return from handler resumes kernel thread or user process
- Faster version (textbook)
 - Interrupt handler returns to saved state in TCB
 - Could be kernel thread or user process

Thread switch

- Save registers (context) of the old thread in the TCB
- Move old thread's TCB to Ready List or to a Waiting List
- Select a new thread from the Ready List
- Restores new thread's registers from TCB to processor
- Put new thread's TCB in the Running List
- return control to the new thread (IRET)

Thread switch - overhead

- Due to registers save and restore
- Due to TCB queues management
- **Memory cache invalidation**
 - link to the computer architecture class
- **Induced operations on the memory manager**
 - Address exceptions
 - Page faults
 - MMU invalidation
 - Will be discussed later on

Context switch - example

Let us consider a processor with special registers PC & PS, the user-level stack pointer SP, the kernel level stack pointer SP' and general registers R1, R2

The interrupt vector is in memory

The system uses a single kernel stack (shared for all threads)

When receives an interrupt, the processor:

- Sets kernel mode;
- Disable interrupts;
- Saves PC & PS & SP on the kernel stack
- Loads the new PC & PS from the interrupt vector
- Consequently jumps to the interrupt handler in the kernel

Hardware

The IRET instruction:

- Enable interrupts;
- Sets user mode;
- Restores PC, PS & SP from the kernel stack; (consequently jumps back to the address at which the RUNNING thread had been interrupted in the past)

The interrupt handler:

- First saves the general registers on the kernel stack
- at the end restores the general registers from the kernel stack and executes IRET

Software

Context switch – example 1

Hyp. A): thread T1 invokes a system call. At the end it remains in RUNNING state

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

kernel SP	0FFF
-----------	------

Context switch - example

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF
----------------	------

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF
----------------	------

Context switch - example

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

3) after temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

Context switch - example

4) at the end of the primitive (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

5) during extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5100
PS	AA45
SP	1002
R1	4500
R2	CD31

Context switch - example

5) during extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5100
PS	AA45
SP	1002
R1	4500
R2	CD31

6) at the end of IRET (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

Context switch – example 2

Hyp. B): thread T1 invokes a system call that switches T2 in RUNNING state

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

address	5000
PS	AA45
interrupt vector	

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel SP	0FFF
-----------	------

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

Context switch - example

1) Initial situation during the execution of SVC instruction (USER MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF
----------------	------

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

address	5000
PS	AA45
interrupt vector	

base kernel SP	0FFF
----------------	------

Context switch - example

2) After interrupt (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	
1003	
1004	

registers	
PC	5000
PS	AA45
SP	1002
R1	4500
R2	CD31

3) After temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

Context switch - example

3) After temporary storage of registers (KERNEL MODE)

TCB T1	
State	Running
PC	????
PS	16F2
SP	????
R1	????
R2	????

TCB T2	
State	Ready
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	1880
1000	16F2
1001	2880
1002	4500
1003	CD31
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

4) After storage of registers of T1 and restore of registers of T2 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	25CC
1003	F012
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

Context switch - example

4) After storage of registers of T1 and restore of registers of T2 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	25CC
1003	F012
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1004
R1	??
R2	??

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	
1003	
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1002
R1	25CC
R2	F012

Context switch - example

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	A12C
1000	16F2
1001	A275
1002	
1003	
1004	

registers	
PC	5000 + ??
PS	AA45
SP	1002
R1	25CC
R2	F012

5) During extraction of IRET at address 5100 (KERNEL MODE)

TCB T1	
State	Waiting
PC	1880
PS	16F2
SP	2880
R1	4500
R2	CD31

TCB T2	
State	Running
PC	A12C
PS	16F2
SP	A275
R1	25CC
R2	F012

kernel stack	
0FFF	
1000	
1001	
1002	
1003	
1004	

registers	
PC	A12C
PS	16F2
SP	A275
R1	4500
R2	CD31